回顾预习录播

React基础预习课程

1. React入门
2. JSX语法
3. 组件
4. 正确使用setState
5. 生命周期
6. 组件复合
7. redux
8. react-redux
9. react-router
10. PureComponent
11. vscode常用插件和antd配置

进阶篇

1. 认识Hook
2. 自定义Hook与Hook使用规则
3. Hook API之useMemo与useCallback

基础项目入门篇

1. 初步创建页面
2. 创建底部导航

React组件化

课堂目标

掌握组件化开发中多种实现技术

1. 掌握context，跨层级传递
2. 掌握高阶组件
3. 了解组件化概念，能设计并实现自己需要的组件
4. 掌握弹窗类实现，掌握传送门使用

资源

1. create-react-app
2. HOC
3. ant design
4. 课堂代码地址

知识点

快速开始

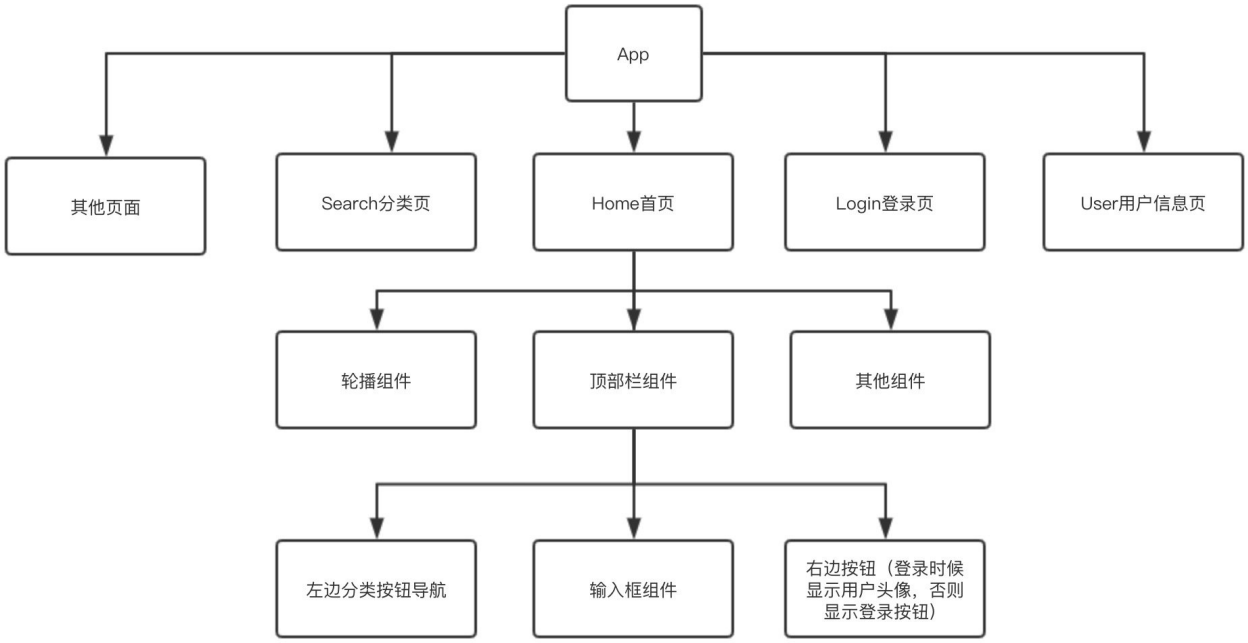
(https://www.html.cn/create-react-app/docs/getting-started/)

```
npx create-react-app lesson1

cd lesson1

yarn start
```

组件化优点

1. 增强代码重用性，提高开发效率
2. 简化调试步骤，提升整个项目的可维护性
3. 便于协同开发
4. 注意点：降低耦合性

组件跨层级通信 - Context



在一个典型的 React 应用中，数据是通过 props 属性自上而下（由父及子）进行传递的，但这种做法对于某些类型的属性而言是极其繁琐的（例如：地区偏好，UI 主题），这些属性是应用程序中许多组件都需要的。Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

React中使用Context实现祖代组件向后代组件跨层级传值。Vue中的provide & inject来源于Context。

Context API

`React.createContext`

创建一个 Context 对象。当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 `Provider` 中读取到当前的 context 值。

`Context.Provider`

Provider 接收一个 `value` 属性，传递给消费组件，允许消费组件订阅 context 的变化。一个 Provider 可以和多个消费组件有对应关系。多个 Provider 也可以嵌套使用，里层的会覆盖外层的数据。

当 Provider 的 `value` 值发生变化时，它内部的所有消费组件都会重新渲染。Provider 及其内部 consumer 组件都不受制于 `shouldComponentUpdate` 函数，因此当 consumer 组件在其祖先组件退出更新的情况下也能更新。

`Class.contextType`

挂载在 class 上的 `contextType` 属性会被重赋值为一个由 `React.createContext()` 创建的 Context 对象。这能让你使用 `this.context` 来消费最近 Context 上的那个值。你可以在任何生命周期中访问到它，包括 render 函数中。

> 你只通过该 API 订阅单一 context。

`Context.Consumer`

这里，React 组件也可以订阅到 context 变更。这能让你在函数式组件中完成订阅 context。

这个函数接收当前的 context 值，返回一个 React 节点。传递给函数的 `value` 值等同于往上组件树离这个 context 最近的 Provider 提供的 `value` 值。如果没有对应的 Provider，`value` 参数等同于传递给 `createContext()` 的 `defaultValue`。

`useContext`

接收一个 context 对象（`React.createContext` 的返回值）并返回该 context 的当前值。当前的 context 值由上层组件中距离当前组件最近的 `<MyContext.Provider>` 的 `value` prop 决定。只能用在function组件中。

使用Context

创建Context => 获取Provider和Consumer => Provider提供值 => Consumer消费值

范例：共享主题色

```
import React, {Component} from "react";
import ContextTypePage from "./ContextTypePage";
import {ThemeContext, UserContext} from "../Context";
import UseContextPage from "./UseContextPage";
import ConsumerPage from "./ConsumerPage";

export default class ContextPage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: {
        themeColor: "red"
      },
      user: {
        name: "xiaoming"
      }
    };
  }

  changeColor = () => {
    const {themeColor} = this.state.theme;
    this.setState({
      theme: {
        themeColor: themeColor === "red" ? "green" : "red"
      }
    });
```

```
  };

  render() {
    const {theme, user} = this.state;
    return (
      <div>
        <h3>ContextPage</h3>
        <button onClick={this.changeColor}>change color</button>
        <ThemeContext.Provider value={theme}>
          <ContextTypePage />
          <UserContext.Provider value={user}>
            <UseContextPage />
            <ConsumerPage />
          </UserContext.Provider>
        </ThemeContext.Provider>
        <ContextTypePage />
      </div>
    );
  }
}
```

//Context.js

```
import React from "react";

export const ThemeContext = React.createContext({themeColor: "pink"});

export const UserContext = React.createContext();
```

pages/ContextTypePage.js

```
import React, {Component} from "react";
import {ThemeContext} from "../Context";

export default class ContextTypePage extends Component {
  static contextType = ThemeContext;
  render() {
    const {themeColor} = this.context;
    return (
      <div className="border">
        <h3 className={themeColor}>ContextTypePage</h3>
      </div>
    );
  }
}
```

pages/ConsumerPage.js

```
import React, {Component} from "react";
import {ThemeContext, UserContext} from "../Context";

export default class ConsumerPage extends Component {
  render() {
    return (
      <div className="border">
        <ThemeContext.Consumer>
          {themeContext => (
            <>
              <h3 className={themeContext.themeColor}>ConsumerPage</h3>
              <UserContext.Consumer>
                {userContext => <HandleUserContext {...userContext} />}
              </UserContext.Consumer>
            </>
          )}
        </ThemeContext.Consumer>
      </div>
    );
  }
}

function HandleUserContext(userCtx) {
  return <div>{userCtx.name}</div>;
}
```

消费多个Context

```
        <ThemeProvider value={theme}>
          <ContextTypePage />
          <ConsumerPage />

          {/*多个Context */}
          <UserProvider value={user}>
            <MultipleContextsPage />
          </UserProvider>
        </ThemeProvider>
```

如果两个或者更多的 context 值经常被一起使用，那你可能要考虑一下另外创建你自己的渲染组件，以提供这些值。

pages/UseContextPage

```
import React, {useState, useEffect, useContext} from "react";
import {ThemeContext, UserContext} from "../Context";

export default function UseContextPage(props) {
  const themeContext = useContext(ThemeContext);
  const {themeColor} = themeContext;
  const userContext = useContext(UserContext);
  return (
    <div className="border">
      <h3 className={themeColor}>UseContextPage</h3>
      <p>{userContext.name}</p>
    </div>
  );
}
```

注意事项

因为 context 会使用参考标识（reference identity）来决定何时进行渲染，这里可能会有一些陷阱，当 provider 的父组件进行重渲染时，可能会在 consumers 组件中触发意外的渲染。举个例子，当每一次 Provider 重渲染时，以下的代码会重渲染所有下面的 consumers 组件，因为 `value` 属性总是被赋值为新的对象：

```
class App extends React.Component {
  render() {
    return (
      <Provider value={{something: 'something'}}>
        <Toolbar />
      </Provider>
    );
  }
}
```

为了防止这种情况，将 value 状态提升到父节点的 state 里：

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: {something: 'something'},
    };
  }

  render() {
    return (
```

```
      <Provider value={this.state.value}>
        <Toolbar />
      </Provider>
    );
  }
}
```

总结

在React的官方文档中，`Context`被归类为高级部分(Advanced)，属于React的高级API，建议不要滥用。

后面我们要学习到的react-redux的`<Provider />`，就是通过`Context`提供一个全局态的`store`，路由组件react-router通过`Context`管理路由状态等等。在React组件开发中，如果用好`Context`，可以让你的组件变得强大，而且灵活。

高阶组件-HOC

为了提高组件复用率，可测试性，就要保证组件功能单一性；但是若要满足复杂需求就要扩展功能单一的组件，在React里就有了HOC（Higher-Order Components）的概念。

定义：**高阶组件是参数为组件，返回值为新组件的函数。**

基本使用

```
// HocPage.js
import React, {Component} from "react";

// hoc: 是一个函数，接收一个组件，返回另外一个组件
//这里大写开头的Cmp是指function或者class组件
const foo = Cmp => props => {
  return (
    <div className="border">
      <Cmp {...props} />
    </div>
  );
};

// const foo = Cmp => {
//   return props => {
//     return (
//       <div className="border">
//         <Cmp {...props} />
//       </div>
//     );
//   };
```

```
// };

function Child(props) {
  return <div> Child {props.name}</div>;
}

const Foo = foo(Child);
export default class HocPage extends Component {
  render() {
    return (
      <div>
        <h3>HocPage</h3>
        <Foo name="msg" />
      </div>
    );
  }
}
```

链式调用

```
// HocPage.js
import React, {Component} from "react";

// hoc：是一个函数，接收一个组件，返回另外一个组件
//这里大写开头的Cmp是指function或者class组件
const foo = Cmp => props => {
  return (
    <div className="border">
      <Cmp {...props} />
    </div>
  );
};
const foo2 = Cmp => props => {
  return (
    <div className="greenBorder">
      <Cmp {...props} />
    </div>
  );
};

// const foo = Cmp => {
//   return props => {
//     return (
//       <div className="border">
//         <Cmp {...props} />
//       </div>
//     );
```

```
//   };
// };

function Child(props) {
  return <div> Child {props.name}</div>;
}

const Foo = foo2(foo(foo(Child)));
export default class HocPage extends Component {
  render() {
    return (
      <div>
        <h3>HocPage</h3>
        <Foo name="msg" />
      </div>
    );
  }
}
```

装饰器写法

高阶组件本身是对装饰器模式的应用，自然可以利用ES7中出现的装饰器语法来更优雅的书写代码。

`yarn add @babel/plugin-proposal-decorators`

更新config-overrides.js

```
//配置完成后记得重启下
const { addDecoratorsLegacy } = require("customize-cra");

module.exports = override(
  ...,
  addDecoratorsLegacy()//配置装饰器
);
```

如果vscode对装饰器有warning，vscode设置里加上
`javascript.implicitProjectConfig.experimentalDecorators": true`

```
//HocPage.js
//...
// !装饰器只能用在class上
// 执行顺序从下往上
@foo2
@foo
@foo
class Child extends Component {
```

```
  render() {
    return <div> Child {this.props.name}</div>;
  }
}

// const Foo = foo2(foo(foo(Child)));
export default class HocPage extends Component {
  render() {
    return (
      <div>
        <h3>HocPage</h3>
        {/* <Foo name="msg" /> */}
        <Child />
      </div>
    );
  }
}
```

组件是将 props 转换为 UI，而高阶组件是将组件转换为另一个组件。

HOC 在 React 的第三方库中很常见，例如 React-Redux 的 connect，我们下节课就会学到。

使用HOC的注意事项

高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

- **不要在 render 方法中使用 HOC**

  React 的 diff 算法（称为协调）使用组件标识来确定它是应该更新现有子树还是将其丢弃并挂载新子树。 如果从 `render` 返回的组件与前一个渲染中的组件相同（`===`），则 React 通过将子树与新子树进行区分来递归更新子树。 如果它们不相等，则完全卸载前一个子树。

  ```
  render() {
    // 每次调用 render 函数都会创建一个新的 EnhancedComponent
    // EnhancedComponent1 !== EnhancedComponent2
    const EnhancedComponent = enhance(MyComponent);
    // 这将导致子树每次渲染都会进行卸载，和重新挂载的操作!
    return <EnhancedComponent />;
  }
  ```

  这不仅仅是性能问题 - 重新挂载组件会导致该组件及其所有子组件的状态丢失。

表单组件设计与实现

antd表单使用

实现用户名密码登录，并实现校验。

//FormPage.js

class

```
import React, {Component, useEffect} from "react";
import {Form, Input, Button} from "antd";

const FormItem = Form.Item;

const nameRules = {required: true, message: "请输入姓名！"};
const passworRules = {required: true, message: "请输入密码！"};

export default class AntdFormPage extends Component {
  formRef = React.createRef();

  componentDidMount() {
    this.formRef.current.setFieldsValue({name: "default"});
  }

  onReset = () => {
    this.formRef.current.resetFields();
  };
  onFinish = val => {
    console.log("onFinish", val); //sy-log
  };
  onFinishFailed = val => {
    console.log("onFinishFailed", val); //sy-log
  };

  render() {
    console.log("AntdFormPage render", this.formRef.current); //sy-log
    return (
      <div>
        <h3>AntdFormPage</h3>
        <Form
          ref={this.formRef}
          onFinish={this.onFinish}
          onFinishFailed={this.onFinishFailed}
          onReset={this.onReset}>
          <FormItem label="姓名" name="name" rules={[nameRules]}>
            <Input placeholder="name input placeholder" />
          </FormItem>
          <FormItem label="密码" name="password" rules={[passworRules]}>
            <Input placeholder="password input placeholder" />
          </FormItem>
          <FormItem>
            <Button type="primary" size="large" htmlType="submit">
              Submit
```

```
            </Button>
          </FormItem>
          <FormItem>
            <Button type="default" size="large" htmlType="reset">
              Reset
            </Button>
          </FormItem>
        </Form>
      </div>
    );
  }
}
```

function实现：

注意 `useForm` 是React Hooks的实现，只能用于函数组件。

```
export default function AntdFormPage(props) {
  const [form] = Form.useForm();

  const onFinish = val => {
    console.log("onFinish", val); //sy-log
  };
  const onFinishFailed = val => {
    console.log("onFinishFailed", val); //sy-log
  };
  const onReset = () => {
    form.resetFields();
  };

  useEffect(() => {
    form.setFieldsValue({name: "default"});
  }, []);

  return (
    <Form
      form={form}
      onFinish={onFinish}
      onFinishFailed={onFinishFailed}
      onReset={onReset}>
      <FormItem label="姓名" name="name" rules={[nameRules]}>
        <Input placeholder="name input placeholder" />
      </FormItem>
      <FormItem label="密码" name="password" rules={[passworRules]}>
        <Input placeholder="password input placeholder" />
      </FormItem>
      <FormItem>
        <Button type="primary" size="large" htmlType="submit">
```

```
          Submit
        </Button>
      </FormItem>
      <FormItem>
        <Button type="default" size="large" htmlType="reset">
          Reset
        </Button>
      </FormItem>
    </Form>
  );
}
```

antd3表单组件设计思路

- 表单组件要求实现**数据收集**、**校验**、**提交**等特性，~~可通过高阶组件扩展~~
- ~~高阶组件给表单组件传递一个input组件**包装函数**接管其输入事件并统一管理表单数据~~
- ~~高阶组件给表单组件传递一个**校验函数**使其具备数据校验功能~~

但是antd3的设计有个问题，就是局部变化会引起整体变化，antd4改进了这个问题。

antd4表单组件实现

antd4的表单基于rc-field-form，github源码地址。

安装rc-field-form，`yarn add rc-field-form`。

使用useForm，仅限function：

```
import React, {Component, useEffect} from "react";
// import Form, {Field} from "rc-field-form";
import Form, {Field} from "../components/my-rc-field-form/";
import Input from "../components/Input";

const nameRules = {required: true, message: "请输入姓名！"};
const passworRules = {required: true, message: "请输入密码！"};

export default function MyRCFieldForm(props) {
  const [form] = Form.useForm();

  const onFinish = val => {
    console.log("onFinish", val); //sy-log
  };

  // 表单校验失败执行
  const onFinishFailed = val => {
    console.log("onFinishFailed", val); //sy-log
  };
```

```
  useEffect(() => {
    console.log("form", form); //sy-log
    form.setFieldsValue({username: "default"});
  }, []);

  return (
    <div>
      <h3>MyRCFieldForm</h3>
      <Form form={form} onFinish={onFinish} onFinishFailed={onFinishFailed}>
        <Field name="username" rules={[nameRules]}>
          <Input placeholder="input UR Username" />
        </Field>
        <Field name="password" rules={[passworRules]}>
          <Input placeholder="input UR Password" />
        </Field>
        <button>Submit</button>
      </Form>
    </div>
  );
}
```

class实现：

```
export default class MyRCFieldForm extends Component {
  formRef = React.createRef();
  componentDidMount() {
    console.log("form", this.formRef.current); //sy-log
    this.formRef.current.setFieldsValue({username: "default"});
  }

  onFinish = val => {
    console.log("onFinish", val); //sy-log
  };

  // 表单校验失败执行
  onFinishFailed = val => {
    console.log("onFinishFailed", val); //sy-log
  };
  render() {
    return (
      <div>
        <h3>MyRCFieldForm</h3>
        <Form
          ref={this.formRef}
          onFinish={this.onFinish}
          onFinishFailed={this.onFinishFailed}>
```

```
          <Field name="username" rules={[nameRules]}>
            <Input placeholder="Username" />
          </Field>
          <Field name="password" rules={[passworRules]}>
            <Input placeholder="Password" />
          </Field>
          <button>Submit</button>
        </Form>
      </div>
    );
  }
}
```

实现my-rc-field-form

实现Form/index

```
import React from "react";
import _Form from "./Form";
import Field from "./Field";
import useForm from "./useForm";

const Form = React.forwardRef(_Form);
Form.Field = Field;
Form.useForm = useForm;

export {Field, useForm};
export default Form;
```

实现Form

```
import React from "react";
import useForm from "./useForm";
import FieldContext from "./FieldContext";

export default function Form({children, onFinish, onFinishFailed, form}, ref)
{
  const [formInstance] = useForm(form);

  React.useImperativeHandle(ref, () => formInstance);

  formInstance.setCallback({
    onFinish,
    onFinishFailed
  });
```

```
  return (
    <form
      onSubmit={event => {
        event.preventDefault();
        event.stopPropagation();
        formInstance.submit();
      }}>
      <FieldContext.Provider value={formInstance}>
        {children}
      </FieldContext.Provider>
    </form>
  );
}
```

实现FieldContext

```
import React from "react";

const warnFunc = () => {
  console.log("----------err--------"); //sy-log
};

const FieldContext = React.createContext({
  registerField: warnFunc,
  setFieldsValue: warnFunc,
  getFieldValue: warnFunc,
  getFieldsValue: warnFunc,
  submit: warnFunc
});

export default FieldContext;
```

实现useForm

```
import React from "react";

class FormStore {
  constructor() {
    this.store = {}; //存储数据，比如说username password
    this.fieldEntities = [];
    // callback onFinish onFinishFailed
    this.callbacks = {};
  }

  // 注册
```

```
registerField = entity => {
  this.fieldEntities.push(entity);
  return () => {
    this.fieldEntities = this.fieldEntities.filter(item => item !== entity);
    delete this.store[entity.props.name];
  };
};

setCallback = callback => {
  this.callbacks = {
    ...this.callbacks,
    ...callback
  };
};

// 取数据
getFiledValue = name => {
  return this.store[name];
};
getFiledsValue = () => {
  return this.store;
};

// 设置数据
setFieldsValue = newStore => {
  this.store = {
    ...this.store,
    ...newStore
  };

  this.fieldEntities.forEach(entity => {
    const {name} = entity.props;

    Object.keys(newStore).forEach(key => {
      if (key === name) {
        entity.onStoreChange();
      }
    });
  });
};

validate = () => {
  let err = [];
  // todo
  this.fieldEntities.forEach(entity => {
    const {name, rules} = entity.props;
    let value = this.getFiledValue(name);
    let rule = rules && rules[0];
    if (rule && rule.required && (value === undefined || value === "")) {
```

```
        //  出错
        err.push({
          [name]: rules.message,
          value
        });
      }
    });
    return err;
  };

  submit = () => {
    console.log("this.", this.fieldEntities); //sy-log
    let err = this.validate();
    // 在这里校验 成功的话 执行onFinish , 失败执行onFinishFailed
    const {onFinish, onFinishFailed} = this.callbacks;
    if (err.length === 0) {
      // 成功的话 执行onFinish
      onFinish(this.getFiledsValue());
    } else if (err.length > 0) {
      // , 失败执行onFinishFailed
      onFinishFailed(err);
    }
  };

  getForm = () => {
    return {
      registerField: this.registerField,
      setCallback: this.setCallback,
      submit: this.submit,
      getFiledValue: this.getFiledValue,
      getFiledsValue: this.getFiledsValue,
      setFieldsValue: this.setFieldsValue
    };
  };
}

// 自定义hook
export default function useForm(form) {
  const formRef = React.useRef();
  if (!formRef.current) {
    if (form) {
      formRef.current = form;
    } else {
      // new 一个
      const formStore = new FormStore();
      formRef.current = formStore.getForm();
    }
  }
  return [formRef.current];
```

```
}
```

实现Field

```
import React, {Component} from "react";
import FieldContext from "./FieldContext";

export default class Field extends Component {
  static contextType = FieldContext;

  componentDidMount() {
    const {registerField} = this.context;
    this.cancelRegisterField = registerField(this);
  }

  componentWillUnmount() {
    if (this.cancelRegisterField) {
      this.cancelRegisterField();
    }
  }

  onStoreChange = () => {
    this.forceUpdate();
  };

  getControlled = () => {
    const {name} = this.props;
    const {getFiledValue, setFieldsValue} = this.context;

    return {
      value: getFiledValue(name), //取数据
      onChange: event => {
        // 存数据
        const newValue = event.target.value;
        setFieldsValue({[name]: newValue});
      }
    };
  };

  render() {
    console.log("field render"); //sy-log
    const {children} = this.props;

    const returnChildNode = React.cloneElement(children,
this.getControlled());
    return returnChildNode;
  }
```

```
}
```

实现my-rc-form

```
import React, {Component} from "react";
// import {createForm} from "rc-form";
import createForm from "../components/my-rc-form/";

import Input from "../components/Input";

const nameRules = {required: true, message: "请输入姓名！"};
const passworRules = {required: true, message: "请输入密码！"};

@createForm
class MyRCForm extends Component {
  constructor(props) {
    super(props);
    // this.state = {
    //   username: "",
    //   password: ""
    // };
  }

  componentDidMount() {
    this.props.form.setFieldsValue({username: "default"});
  }

  submit = () => {
    const {getFieldsValue, validateFields} = this.props.form;
    // console.log("submit", getFieldsValue()); //sy-log
    validateFields((err, val) => {
      if (err) {
        console.log("err", err); //sy-log
      } else {
        console.log("校验成功", val); //sy-log
      }
    });
  };

  render() {
    console.log("props", this.props); //sy-log
    // const {username, password} = this.state;
    const {getFieldDecorator} = this.props.form;
    return (
      <div>
        <h3>MyRCForm</h3>
        {getFieldDecorator("username", {rules: [nameRules]})(
```

```
          <Input placeholder="Username" />
        )}
        {getFieldDecorator("password", {rules: [passworRules]})(
          <Input placeholder="Password" />
        )}
        <button onClick={this.submit}>submit</button>
      </div>
    );
  }
}

export default MyRCForm;
```

实现

```
import React, {Component} from "react";

export default function createForm(Cmp) {
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = {};
      this.options = {};
    }

    handleChange = e => {
      const {name, value} = e.target;
      this.setState({[name]: value});
    };

    getFieldDecorator = (field, option) => InputCmp => {
      this.options[field] = option;
      return React.cloneElement(InputCmp, {
        name: field,
        value: this.state[field] || "",
        onChange: this.handleChange
      });
    };
    setFieldsValue = newStore => {
      this.setState(newStore);
    };
    getFieldsValue = () => {
      return this.state;
    };
    validateFields = callback => {
      // 自己想象吧~
    };
```

```
    getForm = () => {
      return {
        form: {
          getFieldDecorator: this.getFieldDecorator,
          setFieldsValue: this.setFieldsValue,
          getFieldsValue: this.getFieldsValue,
          validateFields: this.validateFields
        }
      };
    };
    render() {
      return <Cmp {...this.props} {...this.getForm()} />;
    }
  };
}
```

弹窗类组件设计与实现

设计思路

弹窗类组件的要求弹窗内容在A处声明，却在B处展示。react中相当于弹窗内容看起来被render到一个组件里面去，实际改变的是网页上另一处的DOM结构，这个显然不符合正常逻辑。但是通过使用框架提供的特定API创建组件实例并指定挂载目标仍可完成任务。

```
// 常见用法如下：Dialog在当前组件声明，但是却在body中另一个div中显示
import React, {Component} from "react";
import Dialog from "../conponents/Dialog";

export default class DialogPage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      showDialog: false
    };
  }
  render() {
    const {showDialog} = this.state;
    return (
      <div>
        <h3>DialogPage</h3>
        <button
          onClick={() =>
            this.setState({
              showDialog: !showDialog
            })
          }>
          toggle
```

```
        </button>
        {showDialog && <Dialog />}
      </div>
    );
  }
}
```

具体实现: Portal

传送门，react v16之后出现的portal可以实现内容传送功能。

范例：Dialog组件

```
// Diallog.js
import React, { Component } from "react";
import { createPortal } from "react-dom";

export default class Dialog extends Component {
  constructor(props) {
    super(props);
    const doc = window.document;
    this.node = doc.createElement("div");
    doc.body.appendChild(this.node);
  }
  componentWillUnmount() {
    window.document.body.removeChild(this.node);
  }
  render() {
    const { hideDialog } = this.props;
    return createPortal(
      <div className="dialog">
        {this.props.children}
        {typeof hideDialog === "function" && (
          <button onClick={hideDialog}>关掉弹窗</button>
        )}
      </div>,
      this.node,
    );
  }
}
```

```css
.dialog {
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  line-height: 30px;
  width: 400px;
  height: 300px;
  transform: translate(50%, 50%);
  border: solid 1px gray;
  text-align: center;
}
```

> 总结一下：
>
> Dialog做得事情是通过调用createPortal把要画的东西画在DOM树上另一个角落。

回顾

回顾预习录播
React组件化
- 课堂目标
- 资源
- 知识点
  - 快速开始
  - 组件化优点
  - 组件跨层级通信 - Context
    - Context API
      - `React.createContext`
      - `Context.Provider`
      - `Class.contextType`
      - `Context.Consumer`
      - `useContext`
    - 使用Context
      - pages/ContextTypePage.js
      - pages/ConsumerPage.js
      - 消费多个Context
      - pages/UseContextPage
    - 注意事项
    - 总结
  - 高阶组件-HOC
    - 基本使用
    - 链式调用
    - 装饰器写法

作业

1. 实现高阶组件版本的Form.createForm，基于rc-form，参考课后补充视频，根据视频补充代码。**注意事项，交的代码片段一定要有validate函数的实现！！！**
2. **作业提交图片！！！** 提交核心代码！！！

下节课内容

lesson2：redux